

Конспект лекции

Системы контроля версий файлов (GIT)

Цель и задачи лекции

Цель – изучить принципы работы системы контроля версий файлов GIT.

Задачи:

1. Изучить принципы работы систем контроля версий
2. Изучить основные команды GIT
3. Дать понимание принципам ветвления

План занятия

1. Системы контроля версий
2. Основы GIT
3. Ветвления GIT

Системы контроля версий

Что такое контроль версий, и зачем он вам нужен? Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Для примеров в этой книге мы будем использовать исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

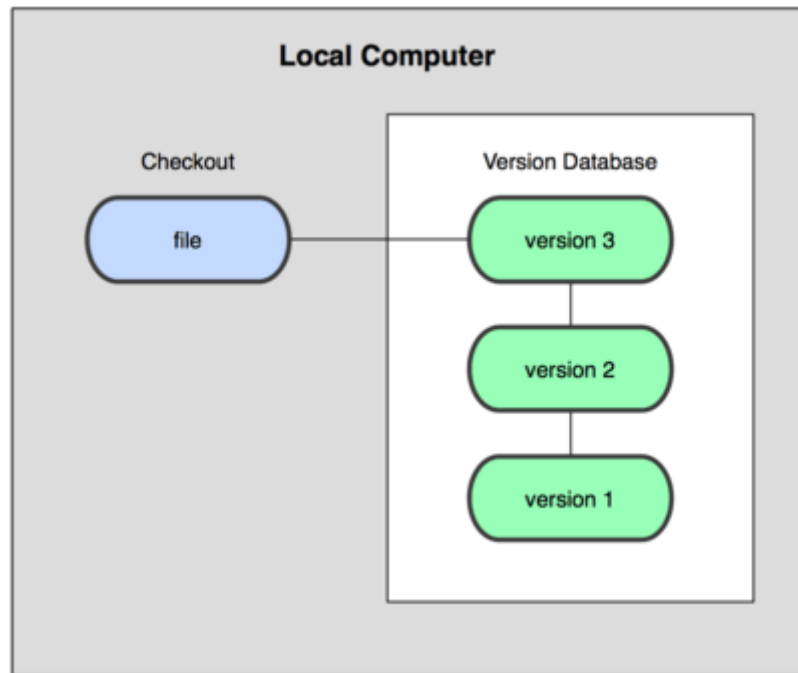
Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета — а этого вам наверняка хочется — то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить. Вдобавок, накладные расходы за всё, что вы получаете, будут очень маленькими.

Локальные системы контроля версий

Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот

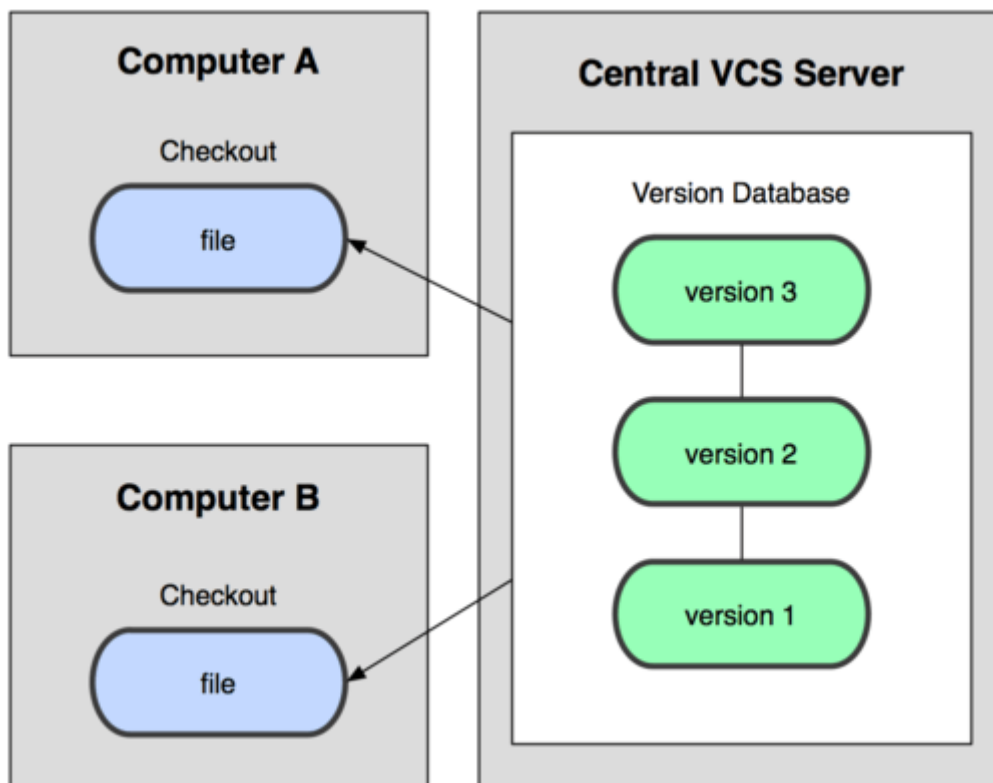
файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы.

Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов.



Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий.

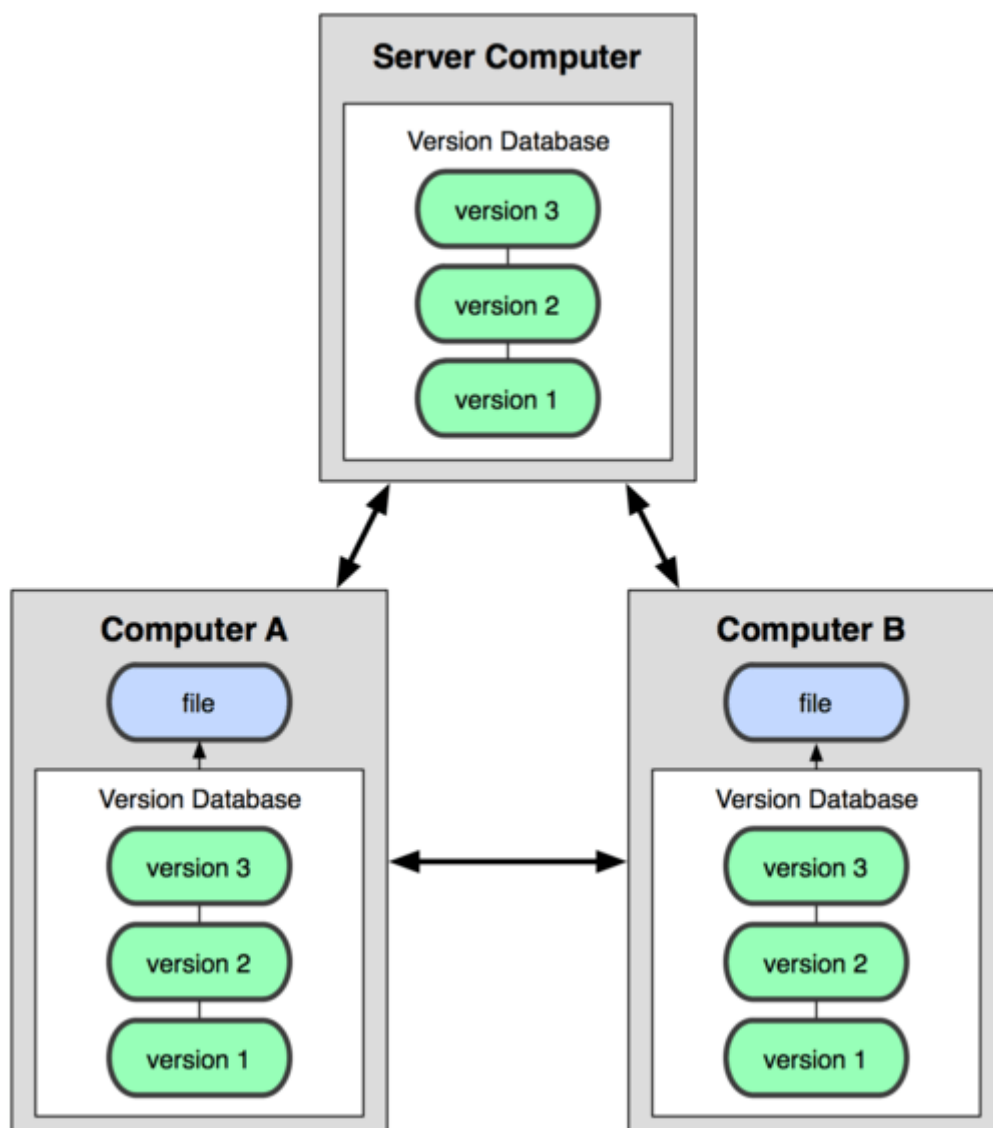


Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных.



Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

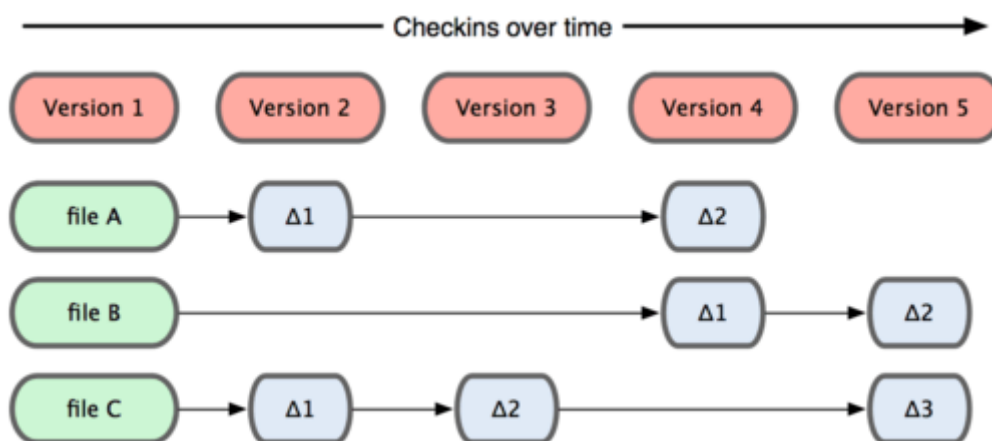
Основы Git

Изучая Git, постарайтесь освободиться от всего, что вы знали о других СКВ, таких как Subversion или Perforce. В Git'e совсем не такие понятия об информации и работе с ней как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git'a.

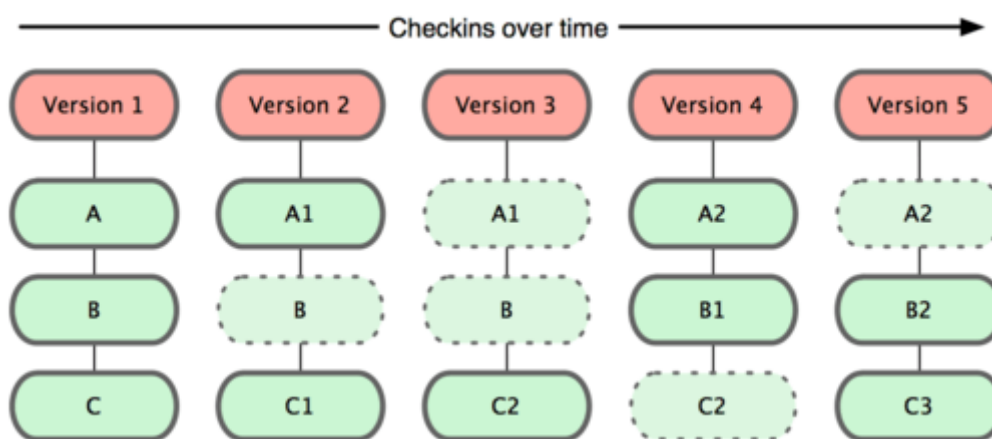
Слепки вместо патчей

Главное отличие Git'a от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к

хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке.



Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 1-5.



Это важное отличие Git'a от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ. В главе 3, коснувшись работы с ветвями в Git'e, мы узнаем, какие преимущества даёт такое понимание данных.

Операции GIT

Для совершения большинства операций в Git'e необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где

практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили Git неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, Git'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

Целостность данных в GIT

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'a и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git'e на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

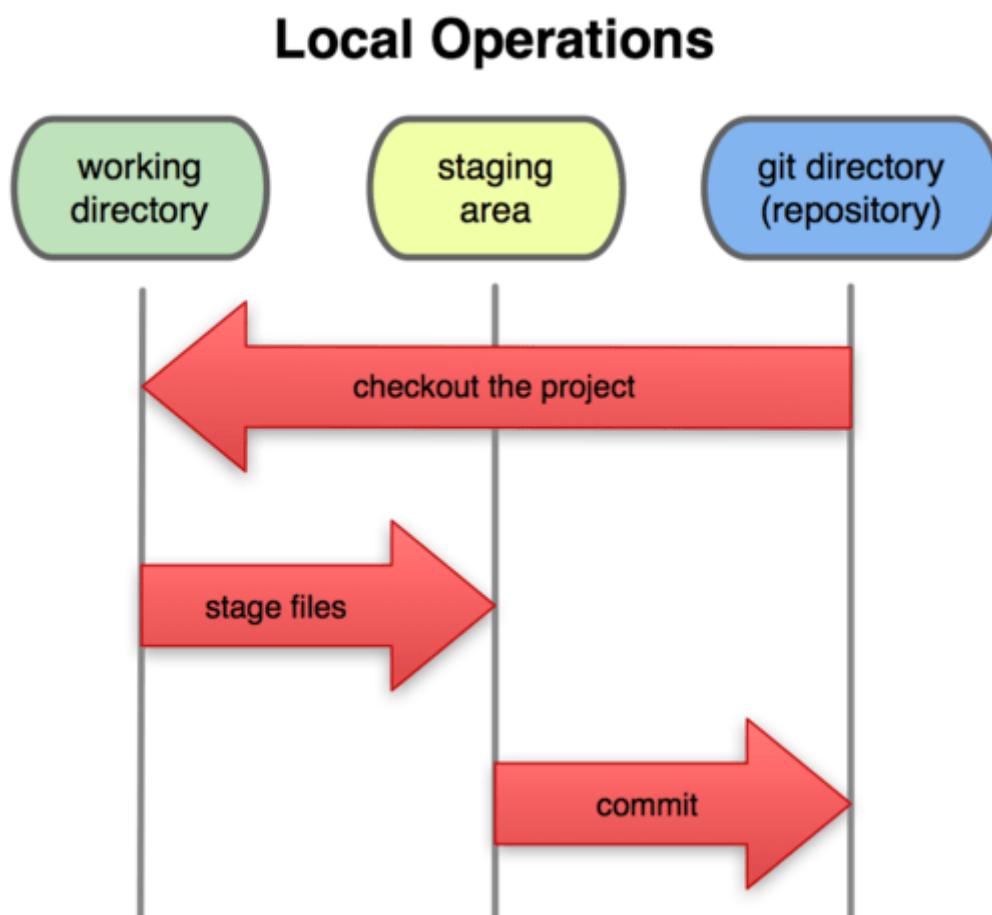
Работая с Git'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Состояния в GIT

Самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. В Git'e файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К

изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части: каталог Git'a (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).



Каталог Git'a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git'a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git'a выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'a на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git'a, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым. В главе 2 вы узнаете больше об этих трёх состояниях и как можно либо воспользоваться этим, либо пропустить стадию подготовки.

Создание репозитория в существующем каталоге

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git` содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. (В главе 9 приведено подробное описание файлов содержащихся в только что созданном вами каталоге `.git`)

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индекслируемые файлы, а затем `commit`:

```
$ git add *.c
```

```
$ git add README
```

```
$ git commit -m 'initial project version'
```

Клонирование существующего репозитория

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется `clone`, а не `checkout`. Это важное отличие — Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете `git clone`. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования.

Клонирование репозитория осуществляется командой `git clone [url]`. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/schacon/grit.git
```

Эта команда создаёт каталог с именем `grit`, инициализирует в нём каталог `.git`, скачивает все данные для этого репозитория и создаёт (`checks out`) рабочую копию последней версии. Если вы зайдёте в новый каталог `grit`, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от `grit`, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван `mygrit`.

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

```
$ git status
```

```
# On branch master nothing to commit, working directory clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла `README`, вы можете выполнить следующее:

```
$ git add README
```

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать файл `.gitignore` с перечислением шаблонов соответствующих таким файлам. Вот пример файла `.gitignore`:

```
$ cat .gitignore
```

*.[oa]

*~

Первая строка предписывает Git'у игнорировать любые файлы заканчивающиеся на .o или .a — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~).

Фиксация изменений

Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Помните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после момента редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли `git status`, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра `-a` в команду `git commit` заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без `git add`

Удаление файлов

Для того чтобы удалить файл из Git'a, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда `git rm`, которая также удаляет файл из вашего рабочего каталога, так что вы в следующий раз не увидите его как “неотслеживаемый”.

Отмена изменений

На любой стадии может возникнуть необходимость что-либо отменить. Здесь мы рассмотрим несколько основных инструментов для отмены произведённых изменений. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git'e, где вы можете потерять свою работу если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда вы делаете коммит слишком рано, забыв добавить какие-то файлы, или напутали с комментарием к коммиту. Если вам хотелось бы сделать этот коммит ещё раз, вы можете выполнить `commit` с опцией `--amend`:

```
$ git commit --amend
```

Эта команда берёт индекс и использует его для коммита. Если после последнего коммита не было никаких изменений (например, вы запустили приведённую команду сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что вы измените, это комментарий к коммиту.

Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Вы можете отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.

Для примера, если после совершения коммита вы осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, вы можете сделать что-то подобное:

```
$ git commit -m 'initial commit'
```

```
$ git add forgotten_file
```

```
$ git commit --amend
```

Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

Работа с удалёнными репозиториями

Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (`push`) и получение (`pull`) данных в и из них тогда, когда нужно обменяться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как отслеживаемые (`tracked`) или нет и прочее. Данный раздел охватывает все перечисленные навыки по управлению удалёнными репозиториями.

Отображение удалённых репозиториев

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду `git remote`. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы клонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, `origin` — это имя по умолчанию, которое Git присваивает серверу, с которого вы клонировали:

```
$ git clone git://github.com/schacon/ticgit.git
```

Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию `-v`:

```
$ git remote -v
```

Добавление удалённых репозиториев

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозиториев, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните `git remote add [сокращение] [url]`:

```
$ git remote
```

```
origin
```

```
$ git remote add pb git://github.com/paulboone/ticgit.git
```

```
$ git remote -v
```

```
origin  git://github.com/schacon/ticgit.git
```

```
pb      git://github.com/paulboone/ticgit.git
```

Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [имя удал. сервера]
```

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

Когда вы клонируете репозиторий, команда `clone` автоматически добавляет этот удалённый репозиторий под именем `origin`. Таким образом, `git`

fetch origin извлекает все наработки, отправленные (push) на этот сервер после того, как вы клонировали его (или получили изменения с помощью fetch). Важно отметить, что команда fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: git push [удал. сервер] [ветка]. Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и, если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён. Вам придётся сначала вытянуть (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

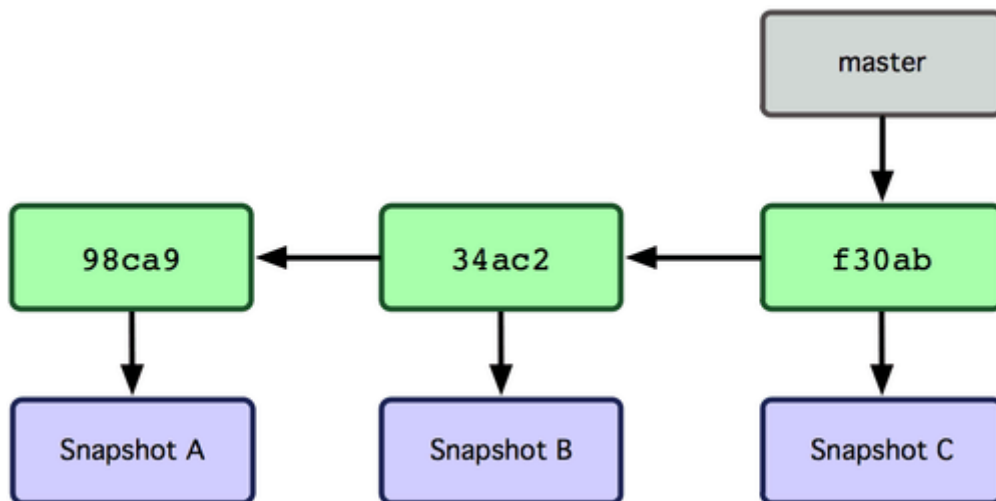
Ветвление в Git

Почти каждая СКВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СКВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

Некоторые говорят, что модель ветвления Git'a это его “killer feature” и она безусловно выделяет Git в СКВ-сообществе. Что же в ней такого особенного? Способ ветвления в Git'e чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение туда-сюда между ветками обычно так же быстрым. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние осуществляется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт вам уникальный мощный инструмент и может буквально изменить то, как вы ведёте разработку.

Ветвления в GIT

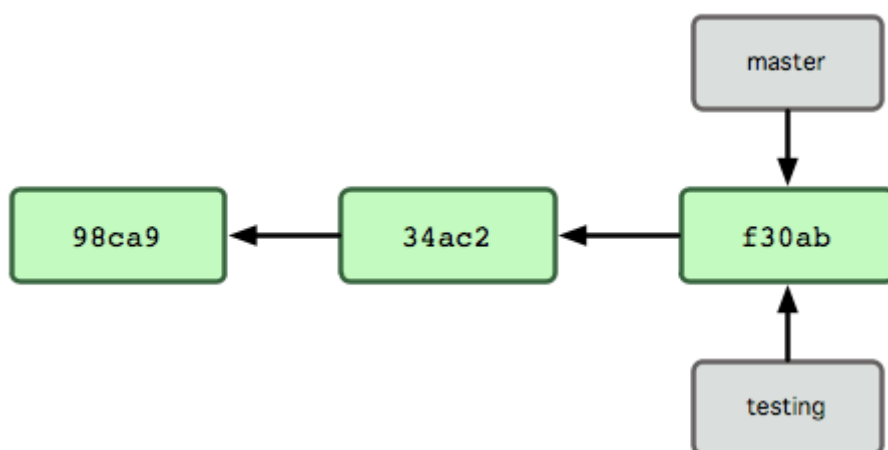
Ветка в Git'e — это просто легковесный подвижный указатель на один из этих коммитов. Ветка по умолчанию в Git'e называется `master`. Когда вы создаёте коммиты на начальном этапе, вам дана ветка `master`, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.



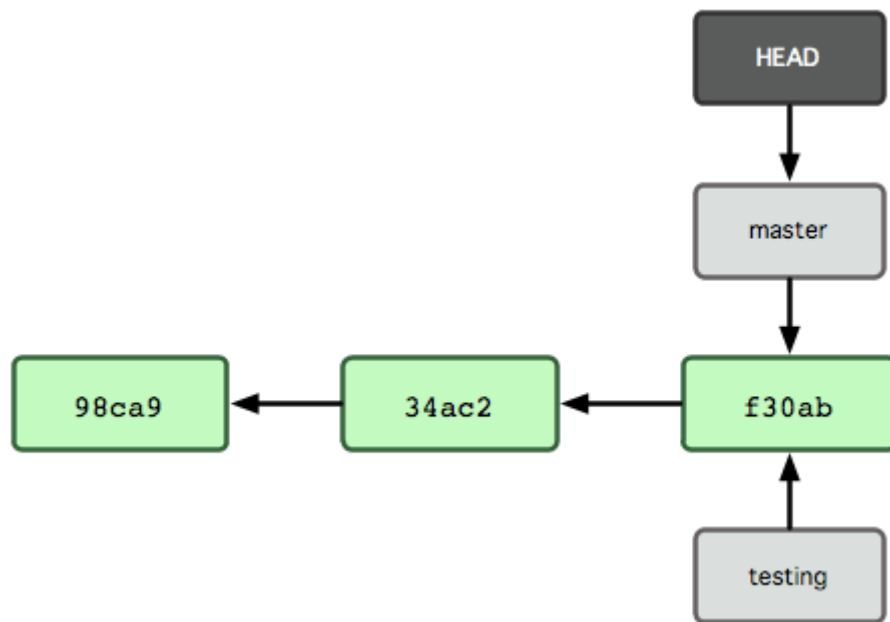
Что произойдёт, если вы создадите новую ветку? Итак, этим вы создадите новый указатель, который можно будет перемещать. Скажем, создадим новую ветку под названием `testing`. Это делается командой `git branch`:

```
$ git branch testing
```

Эта команда создаст новый указатель на тот самый коммит, на котором вы сейчас находитесь.



Откуда Git узнает, на какой ветке вы находитесь в данный момент? Он хранит специальный указатель, который называется `HEAD` (верхушка). Учтите, что это сильно отличается от концепции `HEAD` в других СКВ, таких как Subversion или CVS, к которым вы, возможно, привыкли. В Git'e это указатель на локальную ветку, на которой вы находитесь. В данный момент вы всё ещё на ветке `master`. Команда `git branch` только создала новую ветку, она не переключила вас на неё.



Файл HEAD указывает на текущую ветку. Чтобы перейти на существующую ветку, вам надо выполнить команду `git checkout`. Давайте перейдем на новую ветку `testing`:

```
$ git checkout testing
```

Слияние веток

Для слияния двух разошедшихся веток разработки используя `git merge`:

```
$ git merge branchname
```

Данная команда сливает изменения сделанные в ветке "branchname" в активную(рабочую) ветку. Если присутствуют конфликты -- например один и тот же файл модифицирован разными способами в удаленной и локальной ветках -- то вы будете предупреждены; вывод будет выглядеть след. образом:

```
$ git merge next
```

```
100% (4/4) done
```

```
Auto-merged file.txt
```

```
CONFLICT (content): Merge conflict in file.txt
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Отметки конфликтов останутся в проблемных файлах, и после того, как вы исправите их вручную, вы можете обновить индекс и выполнить `git commit`, как вы обычно это делаете, когда изменяете файл. Если вы просмотрите результирующий коммит, вы увидите, что у него два родителя: один указывает на последний коммит активной ветки, а другой на последний коммит другой ветки.

Исправление конфликтов при слиянии

Когда слияние не происходит автоматически, git оставляет индекс и рабочее дерево в особом состоянии, которое дает всю информацию необходимую чтобы разрешить конфликт.

Файлы с конфликтами отмечаются в индексе особым образом, так что до тех пор пока вы не исправите проблему и не обновите индекс, выполнить git commit не удастся:

```
$ git commit
```

```
file.txt: needs merge
```

Также, git status перечислит эти файлы как "unmerged", а файлы с конфликтами будут иметь добавленные отметки, и выглядеть след.образом:

```
<<<<<<< HEAD:file.txt
```

```
Hello world
```

```
=====
```

```
Goodbye
```

```
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Все что вам нужно это отредактировать файлы, исправить конфликты, а затем выполнить

```
$ git add file.txt
```

```
$ git commit
```

Заметьте, что сообщение-описание коммита уже будет содержать некоторую информацию о слиянии. Обычно вы можете оставить это сообщение-описание, а также можете добавить свои дополнительные комментарии, если пожелаете.

Теперь вы знаете все что вам нужно чтобы выполнить простое слияние. Но git предоставляет больше информации, чтобы помочь разрешить конфликты:

Отменить слияние

Если в процессе слияния и исправления конфликтов, вы застряли и решили сдать все и выбросить все к черту, то вы всегда можете вернуться к состоянию pre-merge (такое же как и было до того как вы запустили слияние) выполнив

```
$ git reset --hard HEAD
```


Если вы уже выполнили коммит после слияния, и вы хотите сбросить его:

```
$ git reset --hard ORIG_HEAD
```

Как бы там ни было, последняя команда может быть опасной в некоторых случаях --никогда не сбрасывайте коммит если этот коммит возможно уже был использован в слиянии в другую ветку, если вы это сделаете то это запутает последующие слияния.

Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда `git branch` делает несколько больше, чем просто создаёт и удаляет ветки. Если вы выполните её без аргументов, то получите простой список имеющихся у вас веток:

```
$ git branch
```

```
iss53
```

```
* master
```

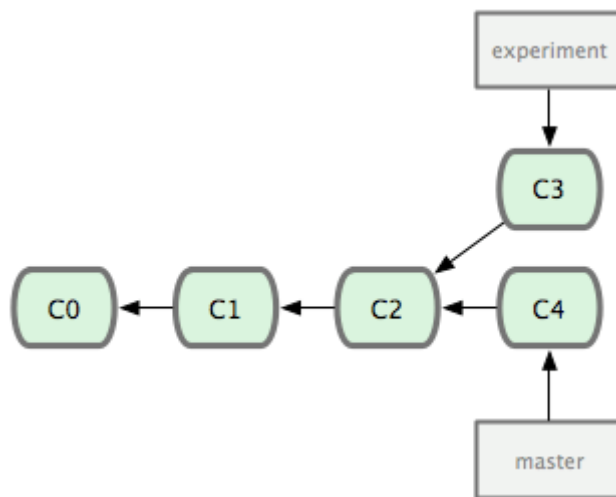
```
testing
```

Обратите внимание на символ `*`, стоящий перед веткой `master`: он указывает на ветку, на которой вы находитесь в настоящий момент. Это означает, что если вы сейчас выполните коммит, ветка `master` переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду `git branch -v`:

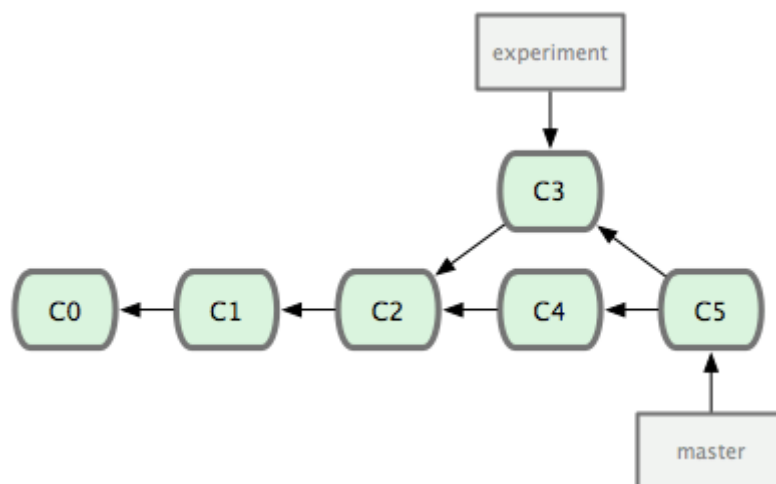
```
$ git branch -v
```

Основы перемещения

Рассмотрим ситуацию, когда мы разделили свою работу на два направления и сделали коммиты на двух разных ветках.



Наиболее простое решение для объединения веток, как мы уже выяснили, команда `merge`. Эта команда выполняет трёхходовое слияние между двумя последними снимками состояний из веток (C3 и C4) и последним общим предком этих двух веток (C2), создавая новый снимок состояния (и коммит).



Однако, есть и другой путь: вы можете взять изменения, представленные в C3, и применить их поверх C4. В Git'е это называется перемещение (rebasing). При помощи команды `rebase` вы можете взять все изменения, которые попали в коммиты на одной из веток, и повторить их на другой.

Для этого примера надо выполнить следующее:

```
$ git checkout experiment
```

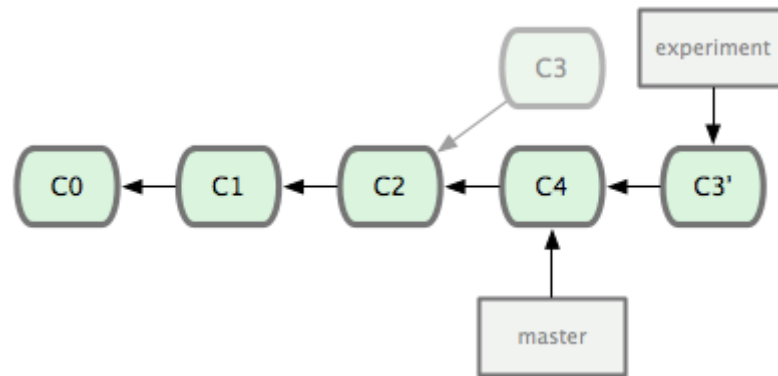
```
$ git rebase master
```

First, rewinding head to replay your work on top of it...

Applying: added staged command

Перемещение работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); для каждого из коммитов в текущей ветке берётся его дельта

и сохраняется во временный файл; текущая ветка устанавливается на тот же коммит, что и ветка, на которую выполняется перемещение; и, наконец, одно за другим применяются все изменения.



Литература и ссылки

1. <https://git-scm.com/book/ru/v1/>
2. Чакон С. Штрауб Б. «Git для профессионального программиста» Питер, 2016 год, 496 стр.

Вопросы для самоконтроля

1. Назовите основные команды GIT.
2. Как реализована работа с ветками в GIT?
3. Для чего применяется версионность в программах?